UML Design and Auto-Generated Code: Issues and Practical Solutions

Ilya Lipkin and Dr. A. Kris Huber
Hill Air Force Base

This article presents issues encountered as well as practical solutions to using the Unified Modeling Language (UML) for design and automatic code generation. Topics presented include general issues found with UML on a real-time systems design project.

This article is based on experience gained during the early history of a project being worked on at Hill Air Force Base, Utah. One of the customer requirements on this project was a specific development tool based on Unified Modeling Language (UML) Version 1.3, namely Rational Rose RealTime (RoseRT).

The project issues and solutions presented in this article are from the real-time control system. The configured software items consist of software design elements expressed in UML from which C++ code can be automatically generated. The observations presented in this article do not necessarily apply to all UML-based development tools, but the authors have made an attempt to raise a few issues of general interest to those involved in similar projects.

Overview of UML

UML is a modeling language developed by Grady Booch, James Rumbaugh, and Ivar Jacobson, and many other contributors to the Object Management Group (OMG). The focus of UML is to model systems using object-oriented concepts and methodology. UML consists of a set of model elements that standardize the design description. These elements include a number of fundamental model elements and modeling concepts, in addition to views that allow designers to examine a design from different perspectives, and diagrams to illustrate the relationships among model elements.

Several views such as Use-Case View, Logical View, Component View, Concurrency View, and Deployment View create a complete description of the system design. Within each view, an organized set of diagrams and other model elements are visible. Diagrams include use-case diagrams, class diagrams, object diagrams, sequence diagrams, collaboration diagrams, statechart diagrams, activity diagrams, component diagrams, and deployment diagrams. Some key primitive model elements are states, transitions, messages, classes, class roles, attributes, and operations [1].

The UML language is complete enough that it offers the exciting ability to allow the creation of auto-generated code that implements the design. The code can be generated from the system description of the model through the use of diagrams and other model elements.

UML Elements for Real-Time Systems Design

Designing real-time systems is challenging. In UML, an *active class* model element was introduced to address this challenge. The purpose of this element was to help simplify both the design and the implementation.

The active class model element consists of a communication structure description and a behavioral description. The communication structure is described using a collaboration diagram that shows the ports through which it sends and receives messages to and from other active classes. The behavior is described using a statechart diagram that shows how the active class acts and reacts to its environment[1]. In other words, the active class is a standalone *capsule* of software that talks to its environment through ports (specified in the structure diagram), and performs *actions* as it transitions through a sequence of states (specified by the statechart diagram).

The characteristics of a run-time system (RTS) object and the UML active class were determined to simplify the process of real-time software design and implementation. In addition, by encapsulating calls to the operating system of the target platform within the RTS, the auto-generated implementation of the UML design can be made largely platform-independent. Real-time application design with UML is discussed in [2].

UML Issues and Practical Solutions

While working on the project, several problems dealing with UML for design and implementation were encountered. Below are some of the issues related to UML design and implementation. These issues, some solutions, and open questions will be presented using small illustrations.

Issue No. 1: State Diagram Clutter

When designing with UML, it is easy for a developer to put too much information into a single diagram. When this happens, information overload will make the design difficult to understand and maintain.

Example

It is possible to describe the behavior of an entire design in a single UML statechart diagram. To demonstrate the potential for diagram clutter, let us consider this simple case: There is a system model that will take user input, button press (an event), and convert it to a textual presentation of the button press to be displayed on screen. An all-UML solution for this simple case is shown in Figure 1.

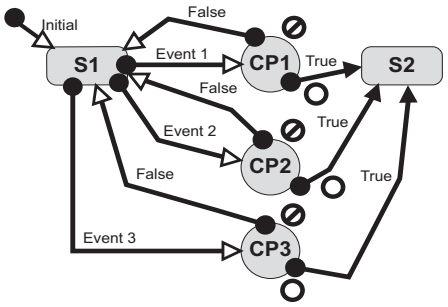
The design consists of a set of three choice points and two states with transitions between them. Each choice point represents a decision to be made when an event-triggered transition has occurred. State S1 represents the entry into the system model and S2 is the final state of the system model.

Since this system model is designed with RoseRT UML elements, the state machine solution is a fully functional implementation and design. For implementation, C++ is auto-generated from the UML design. This simple design solution, although straightforward, is somewhat hard to understand from the statechart diagram in Figure 1. This is because the view of the system is cluttered by transition labels and a large number of UML elements presented in a small visual area.

Practical Solutions

In Figure 2 (see page 14), the solution to the same problem is shown, but utilizing a mixed UML and C++ approach to simplify the model. The *action code* in the transition *Event* would perform the logic implemented by the choice points (circles) in Figure 1, as

Figure 1: *All-UML Solution*



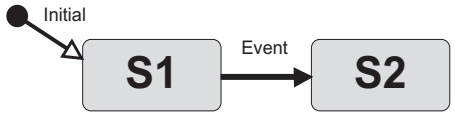


Figure 2: *Combined UML/C++ Solution*

illustrated by the following pseudocode:

```
if(Event1) then
      display string1
else if(Event2) then
      display string2
else if(Event3) then
      display string3
```

This solution moves logic that is straightforward and redundant into the state transition between states S1 and S2. In addition, it also keeps the same overall logic of the UML statechart of Figure 1.

Another benefit to the UML and C++ solution is that the size of the system model can be reduced. The Figure 1 solution produces an implementation consisting of 315 lines of auto-generated code and three lines of *user code* (code that is inserted by the user and copied verbatim by the code generator). On the other hand, the mixed UML/C++ solution in Figure 2 produces an implementation consisting of 215 lines of auto-generated code and 14 lines of user C++ code.

This example demonstrates that the size of the auto-generated code can be reduced through using a mixed UML/C++ design approach. The code efficiency and maintainability also seem to be improved in this case, although this will not be true for all designs. The maintainability of a UML-based software object will depend upon several factors, including the maintenance team's relative proficiency with UML compared to the underlying implementation language.

An alternate method of reducing the number of visual elements on a model diagram is to design hierarchical state machines. If hierarchal statecharts are supported by the design tool, multiple states can be combined to create a clearer, high-level statechart containing fewer states. This approach to reducing visual elements has an additional benefit of increased portability of the design, but may not achieve the implementation efficiency of the combined UML/C++ approach.

Guideline

Although UML is very flexible, some implementation and design considerations should be addressed through conventional means by either manually coding or using system flow diagrams outside of UML [3].

It is important to keep the number of states and transitions to a minimum. As the model complexity increases, understanding of the design can be decreased due to clutter in the design diagrams. The efficiency of the auto-generated implementation of the design can also be reduced. Depending upon the need for implementation efficiency, and the team member familiarity with UML, an appropriate design philosophy should be established for the project.

Issue No. 2: UML Metrics

The authors were not aware of any useful software size metrics that could be used with UML models. Using the source lines of code (SLOC) metric on auto-generated code tended to overemphasize the UML portion of a mixed UML/C++ model (with the UML portion being auto generated and the C++ portion being user code). Metrics were needed both for project estimation and project tracking.

Discussion

Many design rules have been proposed for judging UML design quality [4]. However, it is hard to know how to track effort used on developing software using such metrics. Obviously, SLOC for auto-generated code size could be misleading, but to be feasible, the project tracking and estimation tools often require using a scalar metric like SLOC rather than a vector of metrics. The challenge is to correlate the size metric to the amount of development effort required for design and implementation of the software. Experimentation with the SLOC metric upon auto-generated code seemed to overemphasize the UML portion of a mixed UML/C++ model.

Example

How do we judge the total size of the models of Figures 1 and 2?

Practical Solution

To arrive at the solution presented below, it was necessary to experiment with more than one metric for total size. One of the possible methods is illustrated in the solution of issue No. 1: Measure the auto-generated code separately from the user code that is manually maintained. This approach is adequate for some purposes, but in other situations, a metric is needed that captures the size in a single number.

To remedy shortfalls of the previous approach, a second metric was defined that consists of a single number to represent model size. A utility tool was developed that could traverse the model and provide a total count of several of the key UML model elements. Based on counts of model elements and user code SLOC, this metric could be calculated as a weighted sum. The weights shown in the second column of Table 1 were used to give a size estimate in terms of implementation units (IU).

Using this UML metric, the size of the models in Figures 1 and 2 are calculated from the weighted sum of their model elements to be 107 and 52 IU, respectively. The weights associated with this metric were estimated subjectively based on the RoseRT user interface for each of the model elements. The resulting IU metric has a level of detail similar to SLOC, and the measure may be thought of as a SLOC-equivalent metric. The weights associated with this metric have yet to be fully substantiated due to insufficient historical data for the project. For this reason, project history data is maintained in a raw format so that new weights (and even new metrics) can be applied to the entire project history.

This second metric for estimating the size of a combined UML/C++ model/implementation is similar in spirit to the function points approach. In place of logical or functional elements, UML model elements were given SLOC-equivalent weights.

Guideline

It is best to find a metric that correlates well to the quantity desired. Employ UML design tools that contain rich application programming interfaces (APIs) to facilitate development of utilities that automate the process of metrics calculation. Collecting data in a raw format can facilitate adjustment or customization of the metrics.

Table 1: *Weights and Counts for Key UML Model Elements*

Model Element	Weight (IU)	Counts for Figure 1	Counts for Figure 2
States[2]	6	5	2
Transitions	6	10	2
Capsule Roles	9	1	1
Ports	5	1	1
Attributes	1	0	0
Operations	4	0	0
Signals	2	0	0
User Code	1	3	14
TOTAL (IU)		**107**	**52**

Issue No. 3: Documentation for Graphical Elements

Although UML implements graphical methods of presenting software solutions, it is very challenging to create a self-documenting model. Traditional documentation of the graphical model is still needed.

Discussion

Many developers consider documentation to be the least favorable task of any new design. UML, combined with automatic report generation capability in the tool, allows the possibility of design and implementation to be self-documenting. When using UML for the design task, the designer must create a set of UML diagrams and other graphical information such as states, transitions, and illustrations of the structured relationship among objects. The final product is a model that contains design information in a graphical format that should be suitable for documentation. Ideally if the design is well drawn, then graphical information is actually usable for documentation, and a lot of time and effort savings can be realized.

Example

Figure 1 is an example of a UML model that is not self-documenting. Looking at this figure, several questions can be raised. What do choice points do? What is being compared? What do *true* and *false* labels mean? What do State S1 and State S2 represent? The answer to these questions is that S1 is an initial state and S2 is the final state. The choice points are the logic to set the string to be displayed based on events that occur. *True* and *false* label the condition-dependent transitions from the choice points.

UML tools can produce a sequence diagram from a statechart diagram, as well as generate a number of documentation reports. These typically exclude any user code implementation documentation. Project experience found customization of documentation reports to be non-trivial.

Practical Solution

In the example of Figure 1, a set of supporting documentation is still necessary to accompany a graphical design. Although this seems to be a very obvious observation, the project currently being worked on had omitted to document the graphical design. As a result, the loss in ability to understand and maintain statecharts had increased cost and delayed schedule. In addition, maintainability of the real-time system had been drastically reduced. Although prudent choice of names for the model elements can help make the diagrams more self-documenting, the use of UML use cases and their relationship to the graphical state-chart solutions must be documented rather than assumed to be self-documenting. Software developers must come up with a set of documentation that bridges the link between use cases and their UML graphical representations. The final gap to be filled is the augmentation of documentation generated automatically from UML with manual description for user code and how it fits with design implementation.

Guideline

Although UML presents design visually, comments on the states or any other model elements are still needed. These comments clarify the overall solution and explain some of the design choices of the model diagram [5]. Self-documenting features of UML are still no substitute for design notes or additional supporting documentation on design decisions.

Issue No. 4: Design Portability Between UML Tools (Open Question)

Using custom (i.e., nonstandard) features of UML tools reduces portability of a design to other UML-compliant development tools.

Example

RoseRT had introduced additional concepts into UML to accommodate the needs of the real-time environment, and to allow for a development tool to produce auto-generated code solutions. One of the new concepts to UML introduced by RoseRT was a structure diagram, Figure 3. Structure diagrams are used in RoseRT to link capsules together in a coherent way.

Practical Solution

Unfortunately, there is no straightforward solution for this problem; the only thing that can be used is a set of mitigation strategies. One of those mitigation strategies was the choice of product itself. Although this was done at the program office, the UML implementation chosen was from the primary contributors to the UML standard. Structure diagrams were not part of UML initially, but rather RoseRT's structure diagrams were based on the UML 1.3 collaboration diagram. Another notable difference is the notion of capsules, which are presented in Figures 1 and 2. Capsules are, in essence, UML 1.3 active classes and will be part of UML 2.0 as *structured classes*.

Another strategy is to avoid external API calls in the user code portion of the model, as it will make the design less portable to other UML solutions. Also, the RoseRT UML tools come with a set of custom API calls that can be used to enhance design and development efforts, but it is best to avoid using them if design portability is an issue.

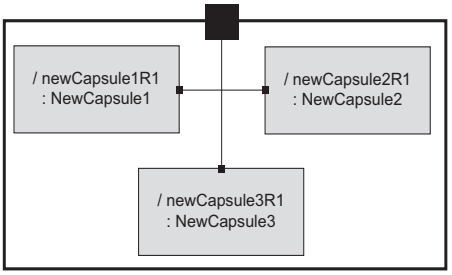


Figure 3: *Structure Diagram*

Guideline

Some of the UML development tools offer specialized controls or functionality that do not exist in other tools. It is a good idea to explore means for design using standardized UML components rather than customized ones. This requires the design team to become familiar with the UML standards to recognize and avoid nonstandard items, thereby providing portability across development tool vendors.

Issue No. 5: Cross Language and Cross Platform Development

Target languages or their compilers can have platform-specific features that defeat the portability of a mixed UML/user-code solution.

Discussion

One of the advantages of UML development is that it is language-neutral, and platform-independent. Using UML, it is possible to disconnect the design from possible issues of implementation that are based on the choice of source code language. In addition, the ability to take UML design as is, and to auto-generate source code from it, allows for the creation of implementation directly from the design [6]. Due to UML being language neutral, the solution will not change if the target language is C/C++, Java, or anything else [7]. In addition, if one of the requirements is to maintain the same code in two languages, it becomes simpler to have bug reports and fixes done in one place rather then two.

The ability to create a design that is platform-independent provides a set of unique opportunities and challenges. Cost may be reduced by enabling a portion of testing to take place on a development platform that does not necessarily include a simulation of the target environment. Multi-platform testing may have a beneficial effect of finding some defects that would otherwise be masked on one of the individual platforms. The challenge of platform-independent design is to ensure that special requirements of the final target environment are

State No., Name	Description	Attributes
Initial	System not running	Timer event set to 30 seconds
Green	On timer time-out event go to(goYellow) Yellow	Timer event set to 45 seconds
Yellow	On timer time-out event go to(goRed) Red	Timer event set to 10 seconds
Red	On timer time-out event go to(goGreen) Green	Timer event set to 30 seconds

Table 2: *State Specification Template*

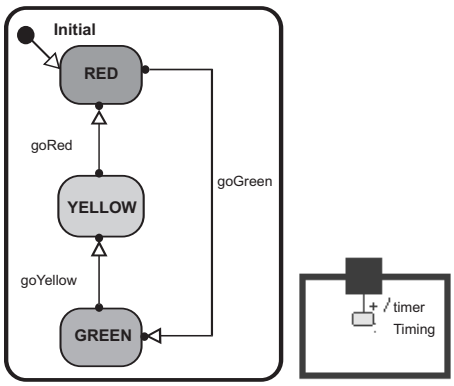


Figure 4: *Working UML Design and Implementation Solution for Traffic Light*

being considered in the design of the user code portion of the UML model.

Example

A traffic light design scenario can be used to demonstrate the ability to have a cross-language and cross-platform development. This small UML design example shows both design and implementation.

Requirements for traffic light are presented below. The traffic light has three colors: red, yellow, and green. To change from green to yellow, an event of time expiring is needed. The events always happen in a fixed sequence of green to yellow to red. It is not allowed to go from green to red, for example. The State Specification Template (Table 2) is used to describe a sequence of state-change events [8]. Timer events are written as generic RoseRT descriptions for each event as "timer.informIn(timeout);" this is the only line of code that is written for the entire design and implementation.

Practical Solution

The UML solution in Figure 4 is completely language- and platform-neutral. There is no user code associated with it that is language- or platform-specific. Therefore it is possible for an auto-code generation engine to translate the UML to a destination language or platform of choice.

Guideline

When implementing UML design, it is best to avoid considerations of how code generation will translate UML to a target language or system platform for the auto-generated portion of the solution. However, when entering user code into mixed UML models, extra care must be taken to avoid a platform-specific syntax in the implementation language.

Conclusion

UML design introduces new and unexplored paradigms that can either simplify the task or make it overly complex. It is important to adhere to proven methods and concepts to utilize what is available. Practical solutions and guidelines related to diagram clutter, UML metrics, documentation, design portability, and cross-language/cross-platform development presented in this article are the tip of an iceberg in a great sea of development.◆

References

1. Sanderfer, Lynn. "How and Why to Use the Unified Modeling Language." CROSSTALK, June 2005 <www.stsc.hill.af.mil/crosstalk/2005/06/0506Sanderfer.html>.
2. Gomaa, Hassan. Designing Concurrent, Distributed, and Real-Time Applications With UML. Addison-Wesley, 2000.
3. Larman, Craig. Applying UML and Patterns: An Introduction to Object-Oriented Design and the Unified Process. 2nd ed. Prentice-Hall, 2002.
4. Wüst, Jürgen. SDMetrics 23 Apr. 2005 <www.sdmetrics.com/LoR.html>.
5. Fowler, Martin. UML Distilled: A Brief Guide to the Standard Object Modeling Language. 3rd ed. Addison-Wesley, 2004.
6. Booch, Grady, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide. 2nd ed. Addison-Wesley, 2005.
7. Satzinger, John W., Robert B. Jackson, and Stephen D. Burd. Object-Oriented Analysis and Design with the Unified Process. Thomson, 2005.
8. Humphrey, Watts S. A Discipline for Software Engineering. Addison-Wesley, 1995.

Notes

1. In the RoseRT tool, active classes are called capsules; the associated collaboration diagrams are called *structure* diagrams.
2. Choice points are considered *pseudostates* that are counted as states when calculating this metric.

About the Authors

Ilya Lipkin is an electronics engineer at the 309th Software Maintenance Group at the Ogden Air Logistics Center, Hill Air Force Base, Utah. His current research interests include artificial intelligence, human knowledge capture and analysis, neural networks, fuzzy logic, user interface design, software engineering, and customer relations management. Lipkin has a Bachelor of Science in computer engineering from the University of Toledo, a Master of Science in computer engineering from the University of Michigan, and is a doctoral candidate at the University of Toledo Business School.

309 SMXG/MXDEE
7278 4th ST BLDG 100
Hill AFB, UT 84056
Phone: (801) 586-4477
Fax: (801) 586-2042
E-mail: ilya.lipkin@hill.af.mil

A. Kris Huber, Ph.D., is an electronics engineer at the 309th Software Maintenance Group at the Ogden Air Logistics Center, Hill Air Force Base, Utah, where he has been working for two years on an embedded software engineering project. Previously, he worked on video compression algorithm research, development, and MPEG-4 standardization for Sorenson Media. His interests are software engineering, computers, and electronic systems. Huber has a Bachelor of Science in electrical engineering from Brigham Young University, and master's and doctorate degrees in electrical engineering from Utah State University.

309 SMXG/MXDEE
7278 4th ST BLDG 100
Hill AFB, UT 84056
Phone: (801) 586-5535
Fax: (801) 586-2042
E-mail: kris.huber@hill.af.mil